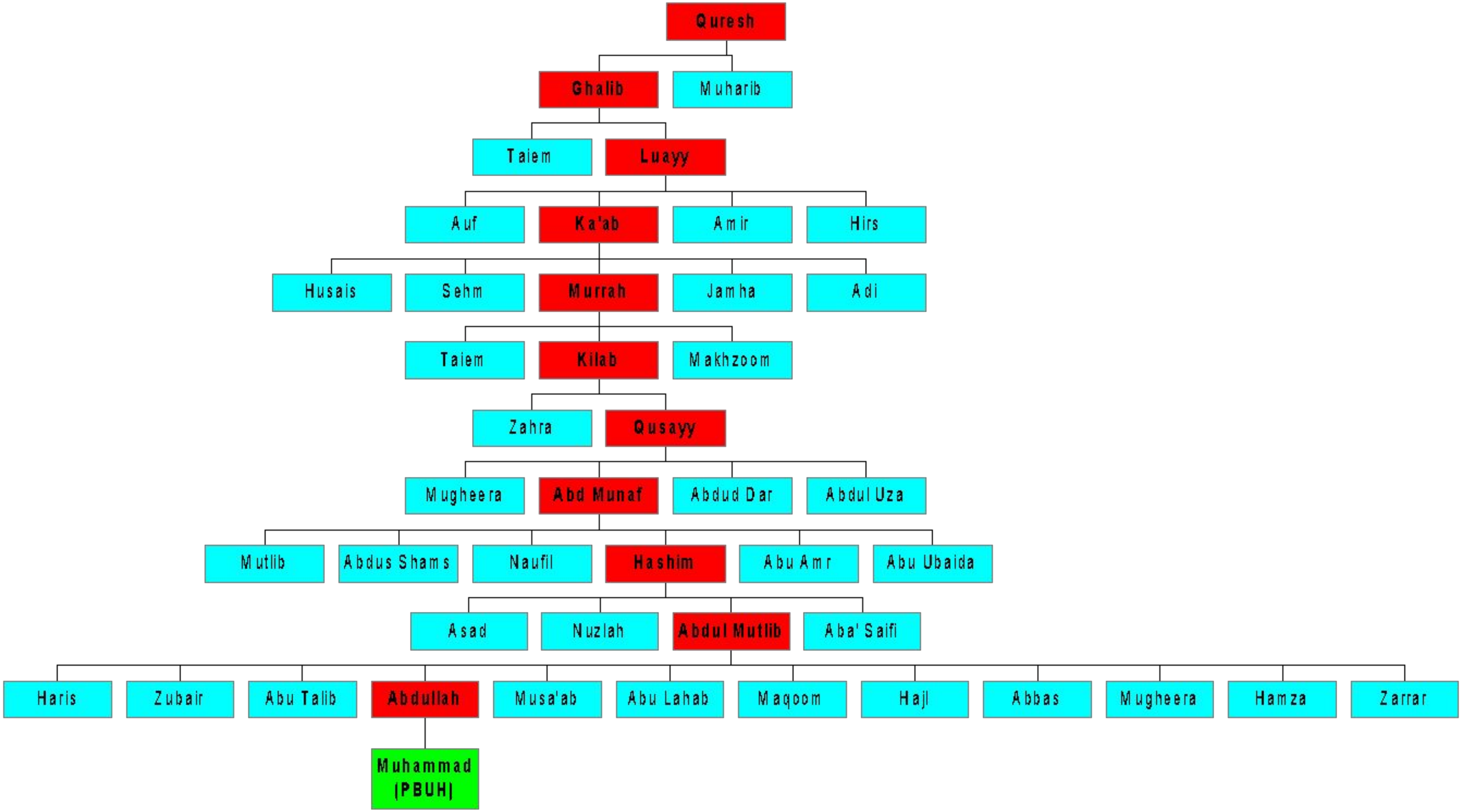
Trees

Family Tree



Quresh

Ghalib
Muharib

Taiem
Luayy

Auf
Ka'ab
Amir
Hirs

Husais
Sehm
Murrah
Jamha
Adi

Taiem
Kilab
Makhzoom

Zahra
Qusayy

Mugheera
Abd Munaf
Abdud Dar
Abdul Uza

Mutlib
Abdus Shams
Naufil
Hashim
Abu Amr
Abu Ubaida

Asad
Nuzlah
Abdul Mutlib
Aba' Saifi

Haris
Zubair
Abu Talib
Abdullah
Musa'ab
Abu Lahab
Maqoom
Hajl
Abbas
Mugheera
Hamza
Zarrar

Muhammad
(PBUH)

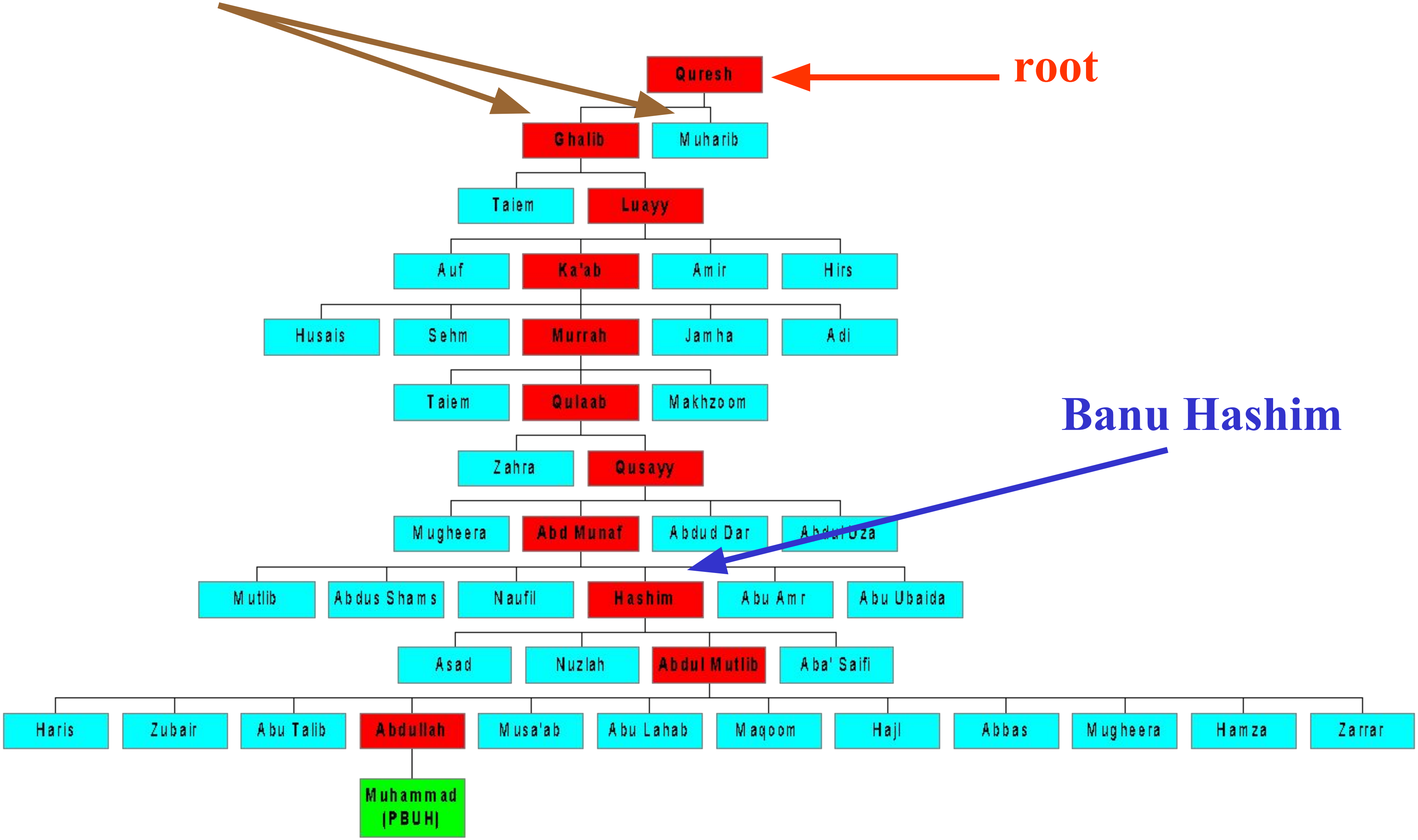
Tree - Definition

A tree is a finite set of one or more nodes such that:

1. There is a specially designated node called the *root.*
2. The remaining nodes are partitioned in $n \geq 0$ disjoint sets T_1, T_2, …, T_n, where each of these sets is a tree.
3. T_1, T_2, …, T_n are called the *sub-trees* of the root.

Recursive Definition

Family Tree



Sub-trees

root

Banu Hashim

Quresh

Ghalib Muharib

Taiem Luayy

Auf Ka'ab Amir Hirs

Husais Sehm Murrah Jamha Adi

Taiem Qulaab Makhzoom

Zahra Qusayy

Mugheera Abd Munaf Abdud Dar Abdul Uza

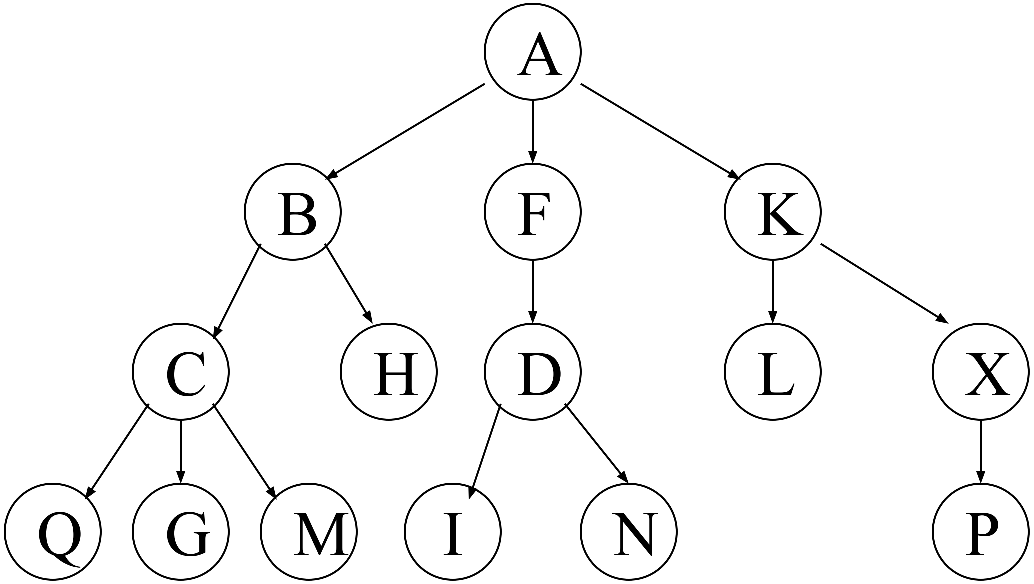
Mutlib Abdus Shams Naufil Hashim Abu Amr Abu Ubaida

Asad Nuzlah Abdul Mutlib Aba' Saifi

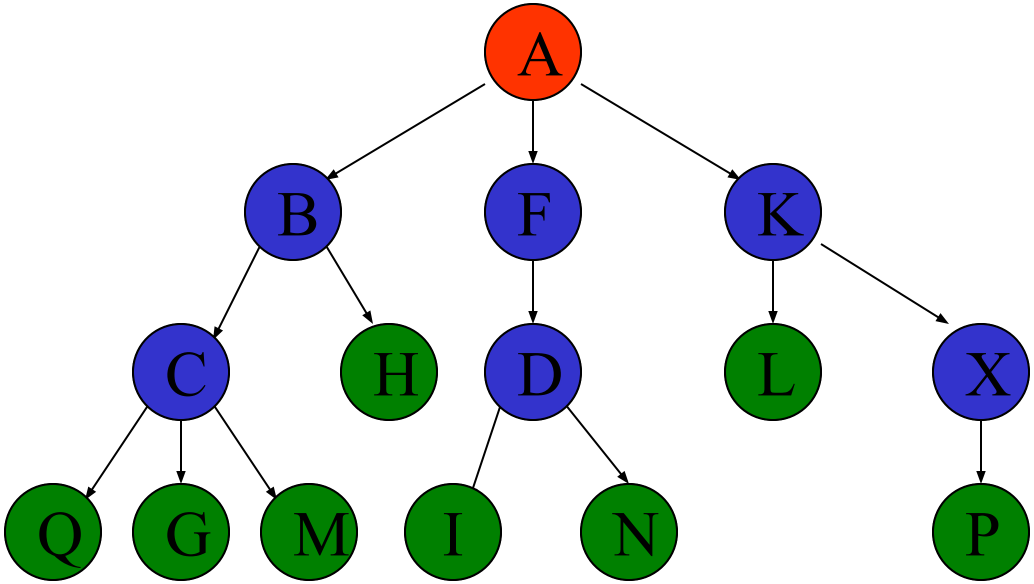
Haris Zubair Abu Talib Abdullah Musa'ab Abu Lahab Maqoom Hajl Abbas Mugheera Hamza Zarrar

Muhammad (PBUH)

Tree



Node Types



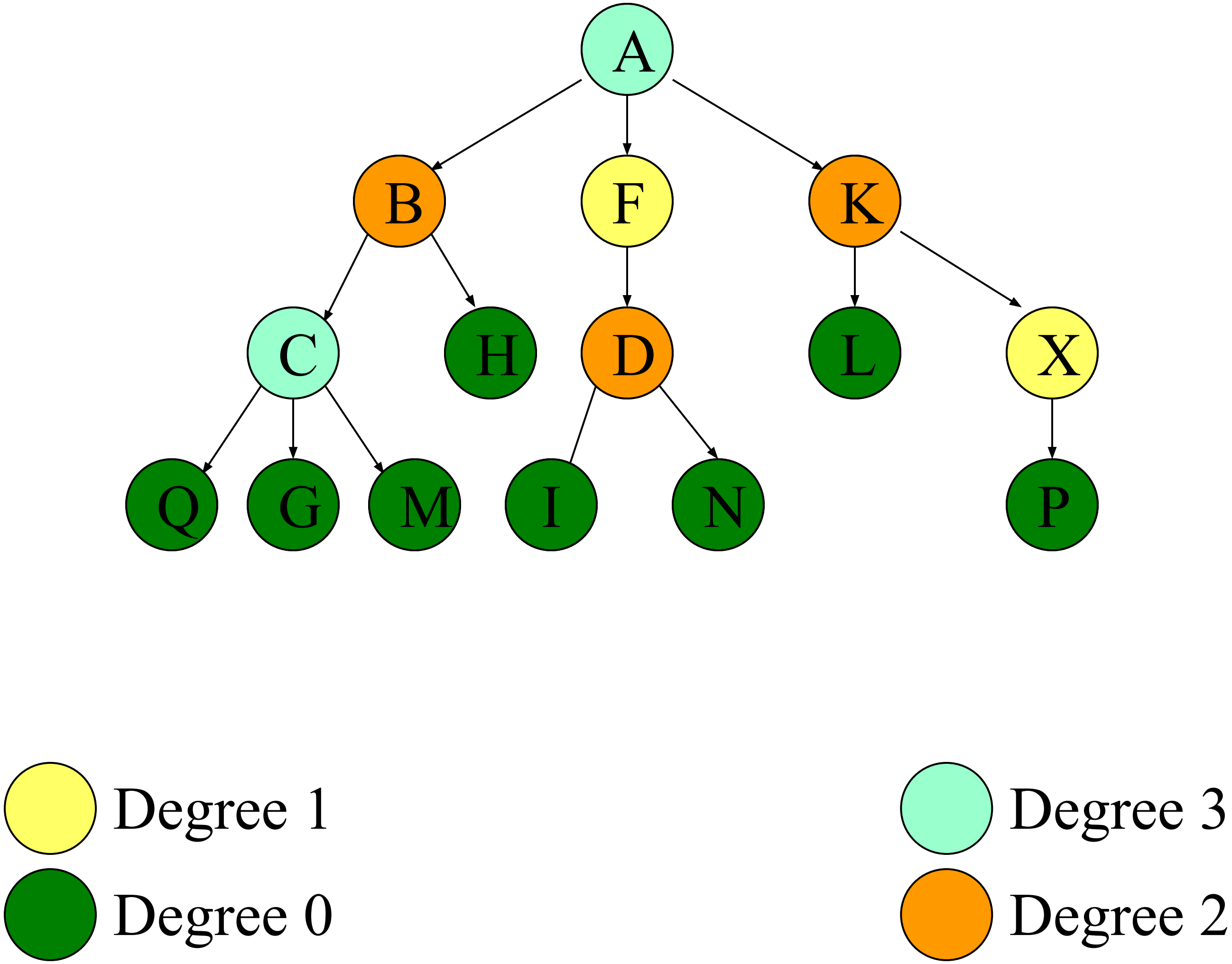
🔴 Root (no parent)

🔵 Intermediate nodes (has a parent and at least one child)

🟢 Leaf nodes (0 children)

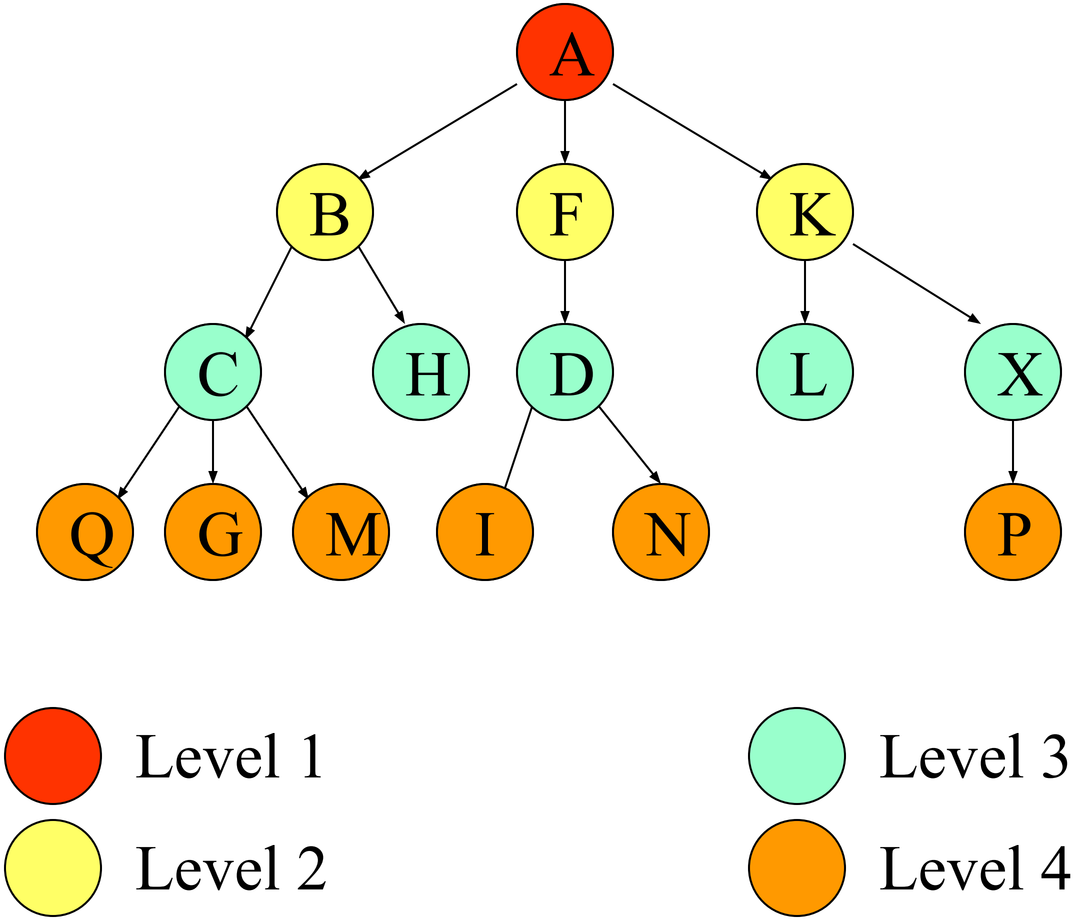
Degree of a Node

Number of Children



Level of a Node

Distance from the root



Height of the Tree = Maximum Level of any Node in the tree

Terminology

- The degree of a node is the number of subtrees of the node.
- The node with degree 0 is a leaf or terminal node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the path from the root to the node.
- The descendant of a node are all the nodes towards the root that has parent of that node and parent’s of that node.

Terminology

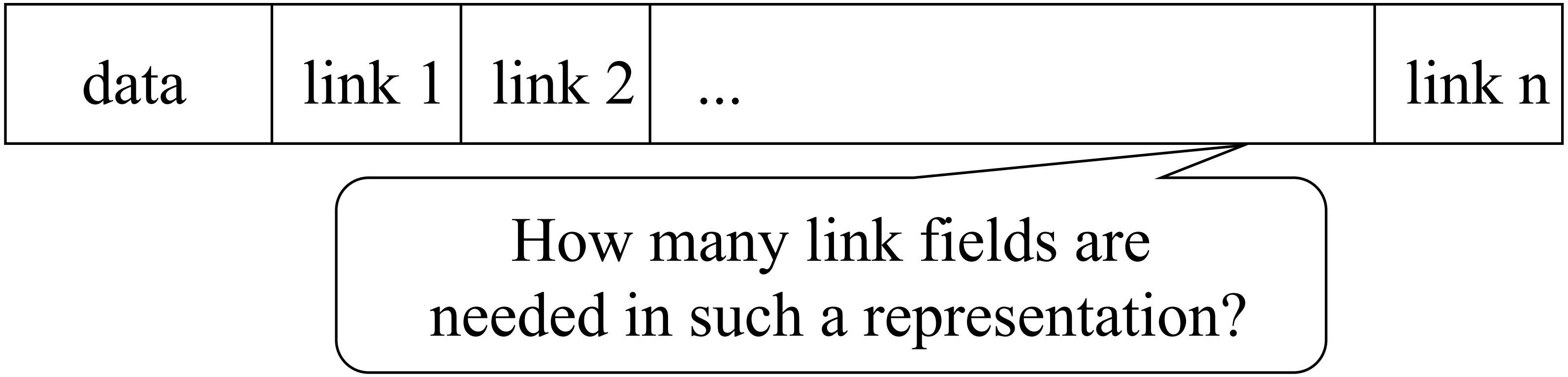
- Height of a tree is the maximum number of Levels in a tree.
- Depth is the reverse of Height.
- Depth of a tree is maximum level of any leaf in the tree.

  Root has maximum height.

  Leaf has maximum Depth.

Representation of Trees

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of sub-trees

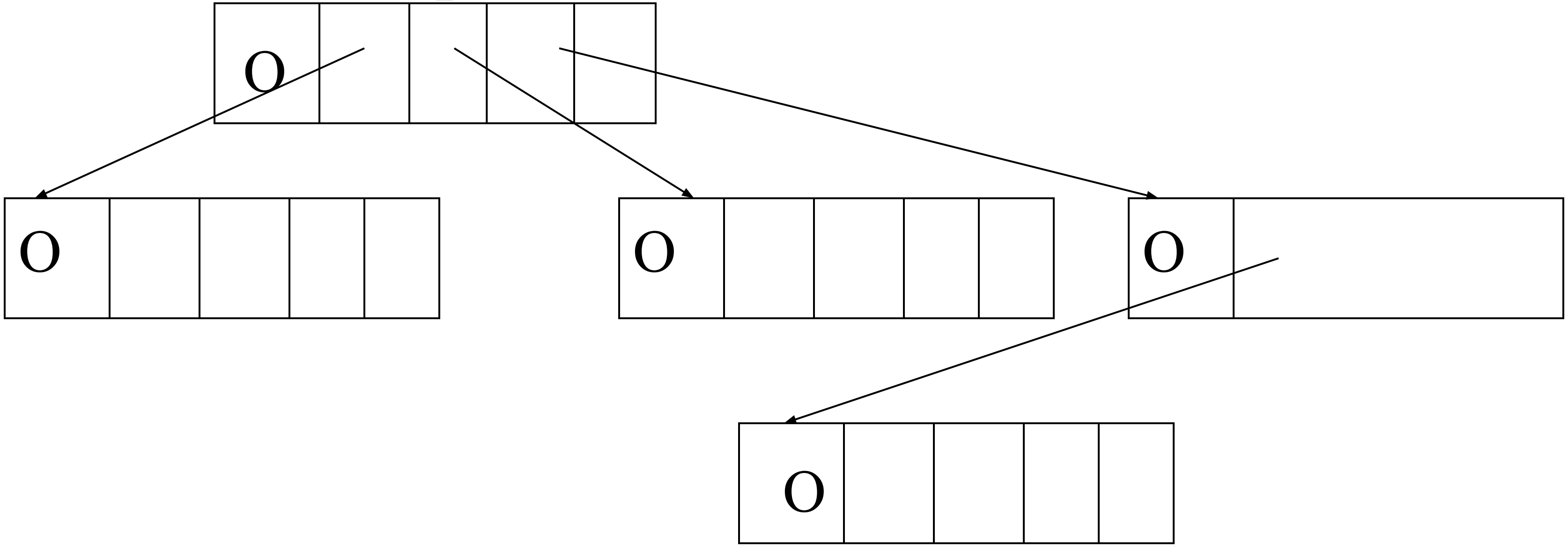


data	link 1	link 2	...	link n

How many link fields are needed in such a representation?

A Tree Node

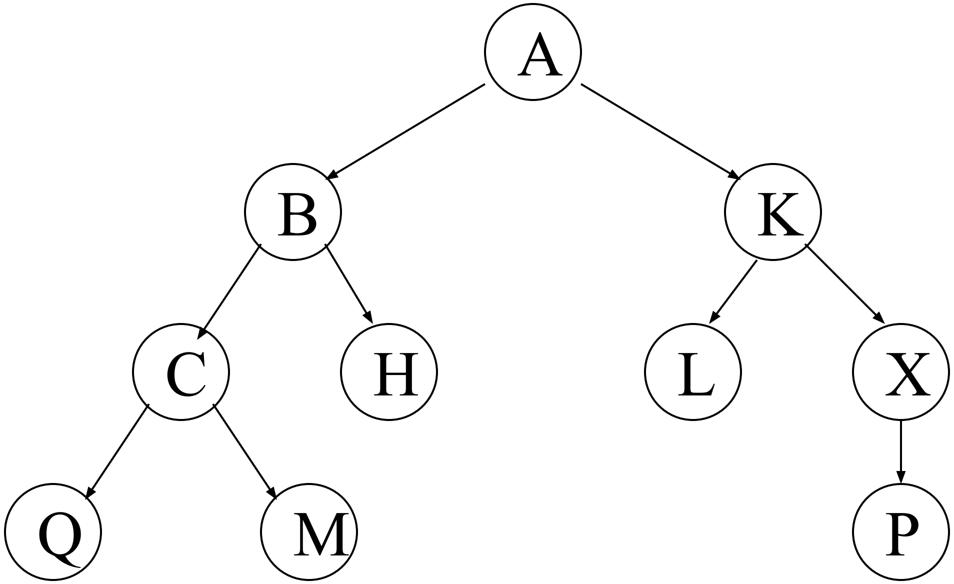
- Every tree node:
  - – object – useful information
  - – children – pointers to its children nodes

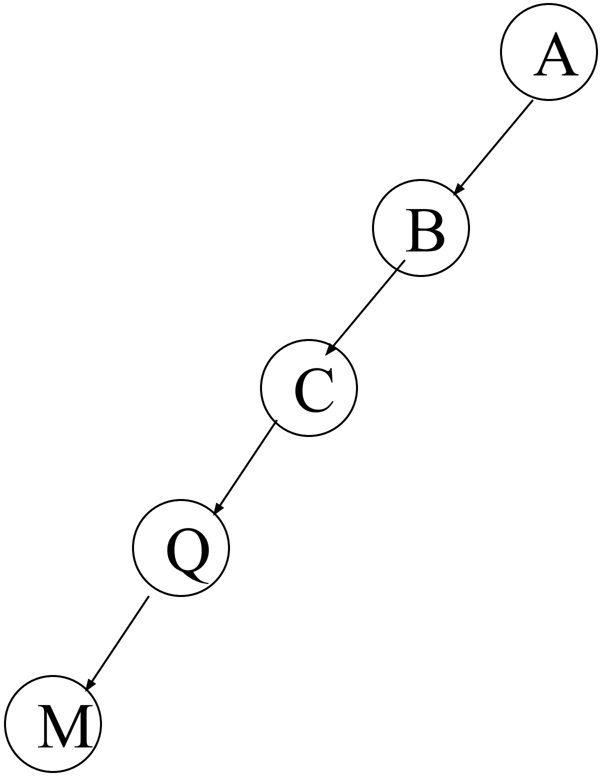


Binary Trees

- A special class of trees: max degree for each node is 2
- Recursive definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.

Examples of Binary Trees





ADT Binary Tree

objects: a finite set of nodes either empty or consisting of a root node, left *BinaryTree*, and right *BinaryTree*.

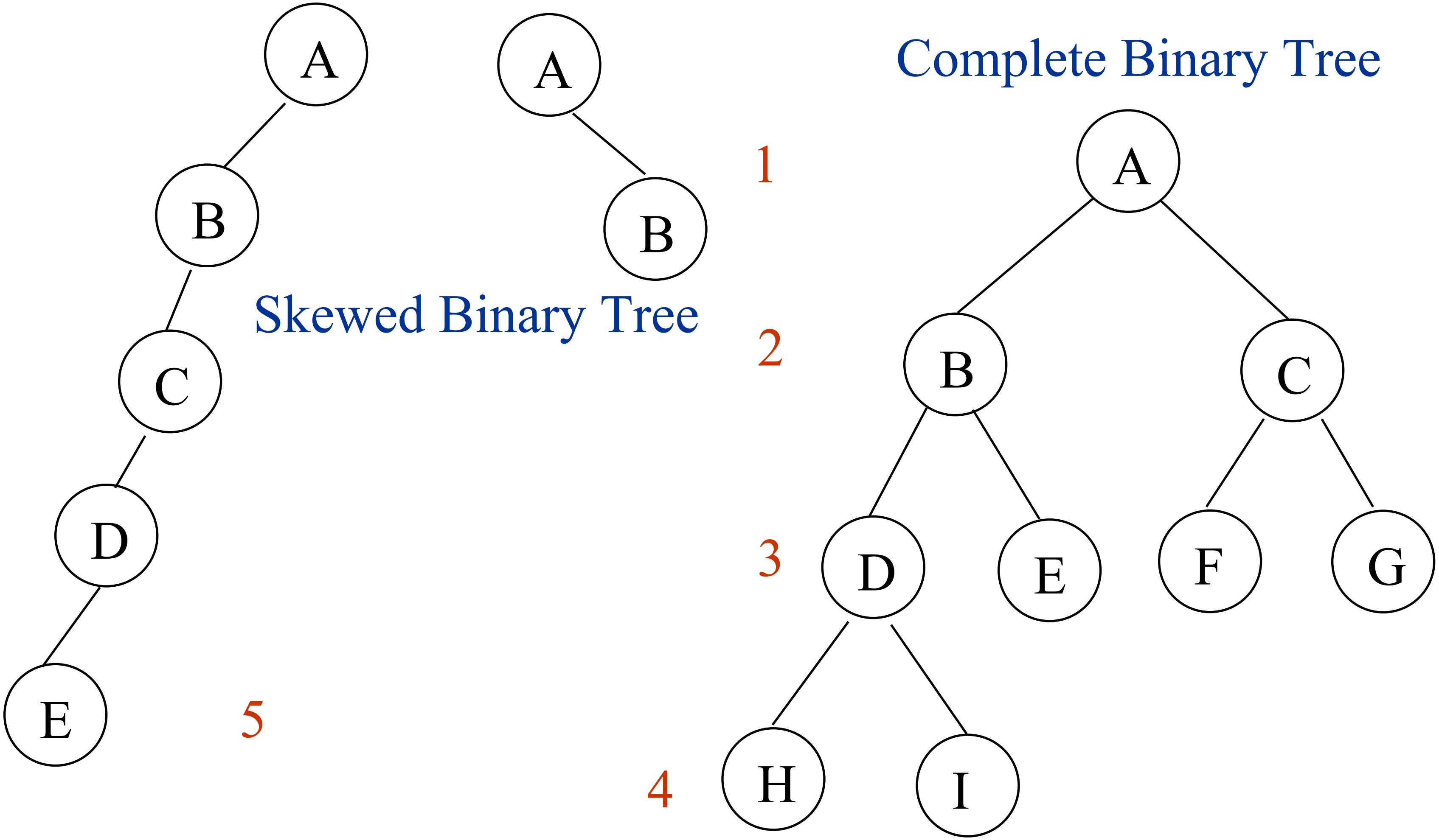
method:

Bintree create()::= creates an empty binary tree

Boolean isEmpty()::= if (**this*==empty binary tree) return *TRUE* else return *FALSE*

```
Bintree leftChild()::= if (IsEmpty()) return error
                       else return the left subtree of *this
element data()::= if (IsEmpty()) return error
                       else return the data in the node of *this
Bintree rightChild()::= if (IsEmpty()) return error
                       else return the right subtree of *this
```

Samples of Binary Trees



Skewed Binary Tree

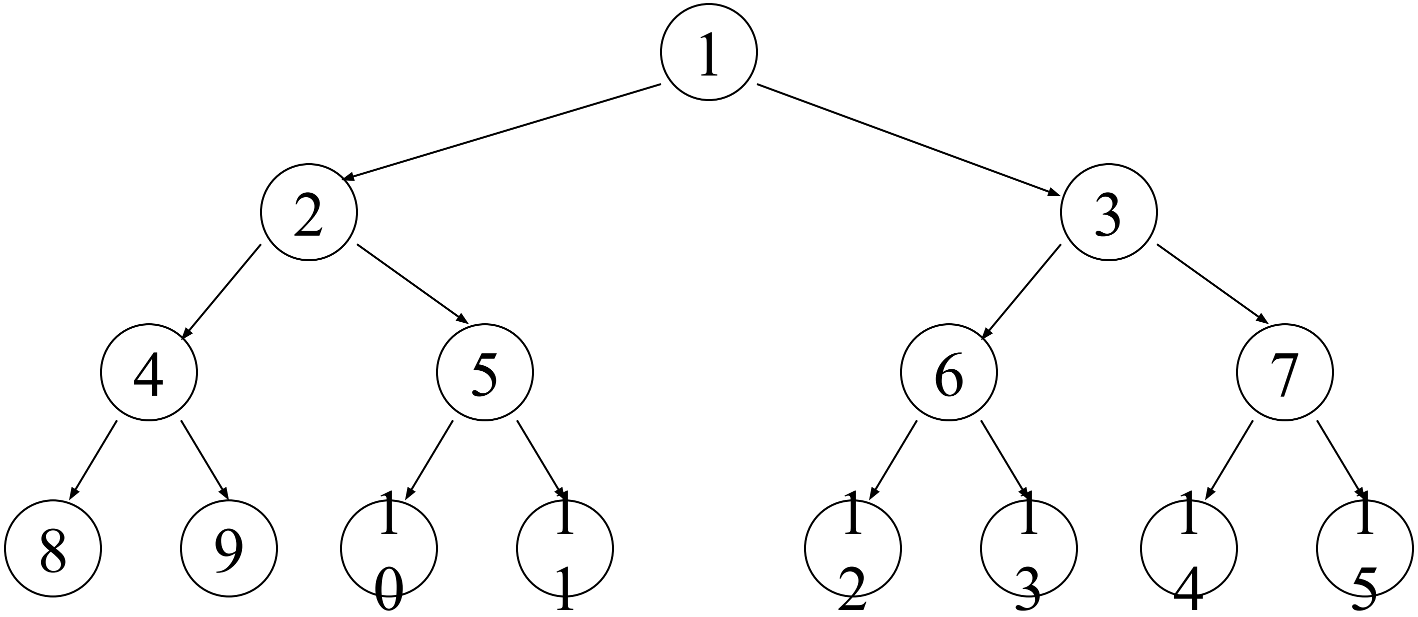
Complete Binary Tree

Properties of Binary Trees

- The maximum number of nodes on level *i* of a binary tree is 2^{i-1} , i>=1
- The maximum number of nodes in a binary tree of height *k* is $2^k – 1$, k>=1

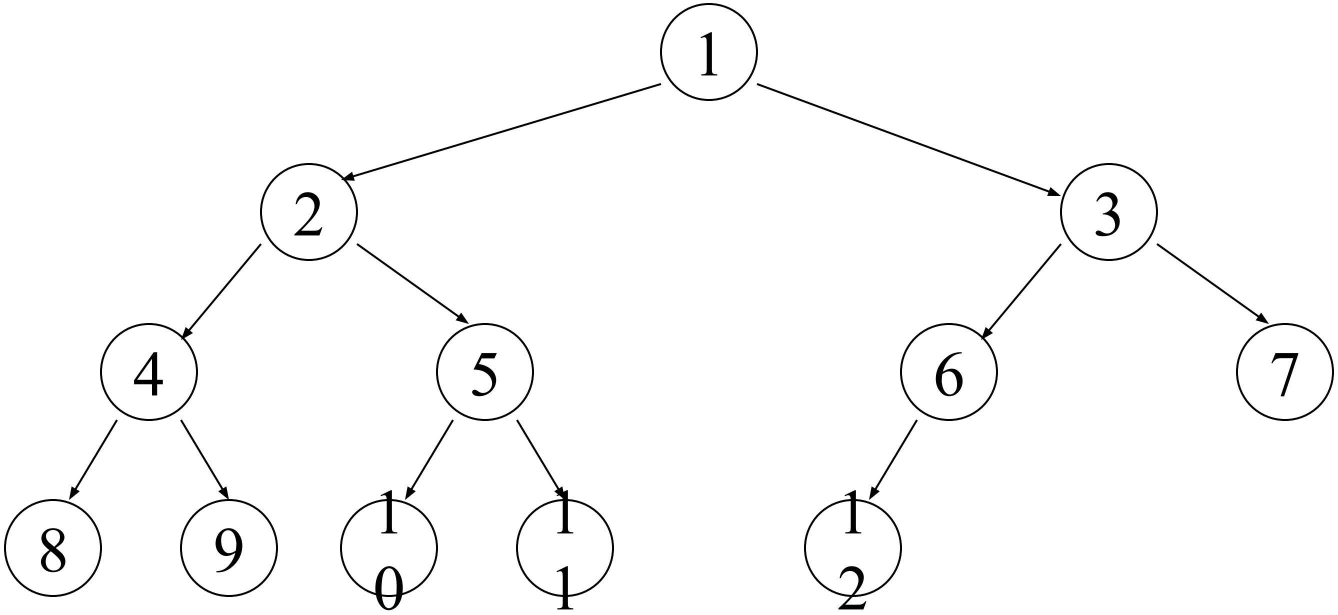
Full Binary Tree

A binary tree of height *k* having *2^k – 1* nodes is called a *full* binary tree



Complete Binary Tree

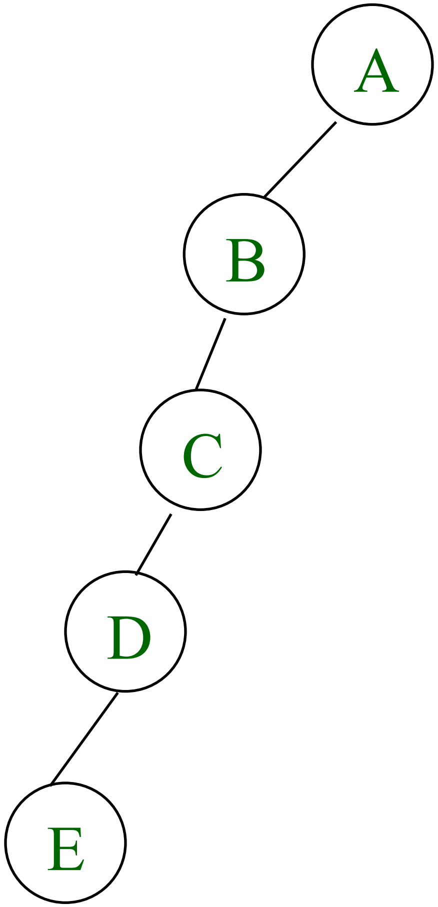
A binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right, is called a *complete* binary tree



Binary Tree Representations

- If a complete binary tree with *n* nodes
  - *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.
  - *leftChild*(*i*) is at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child.
  - *rightChild*(*i*) is at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.

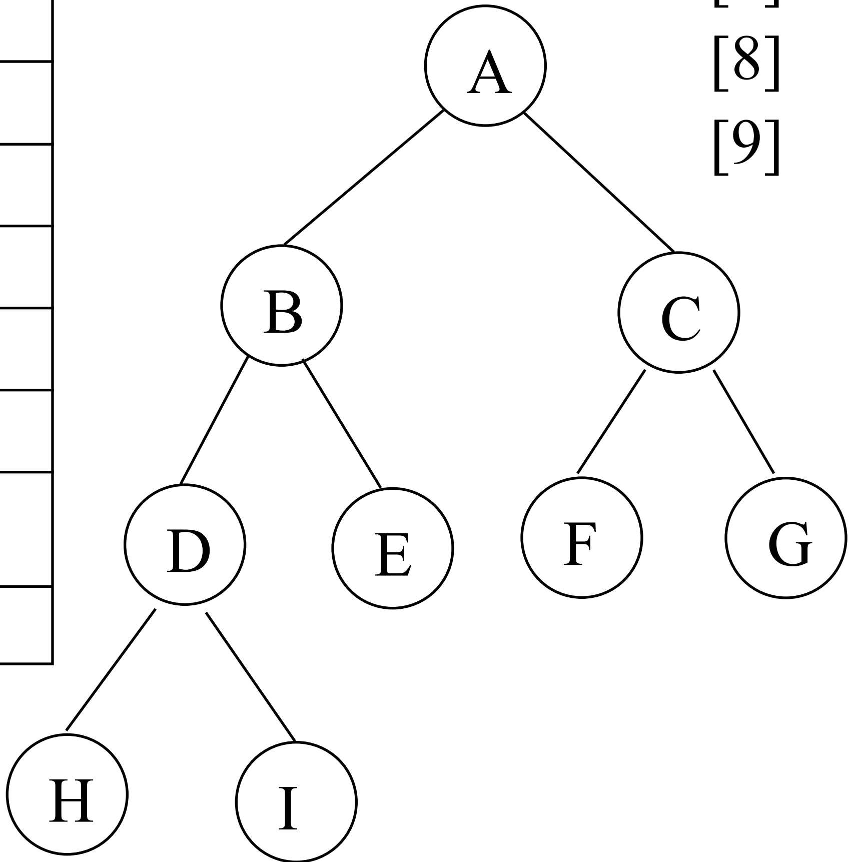
Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

(1) waste space

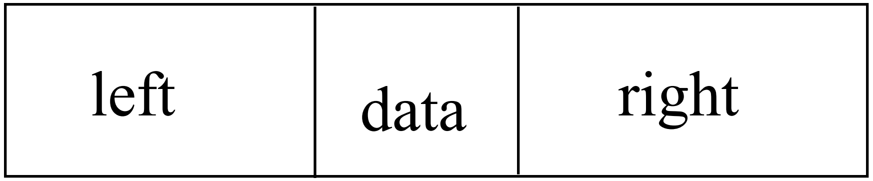
(2) insertion/deletion problem



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

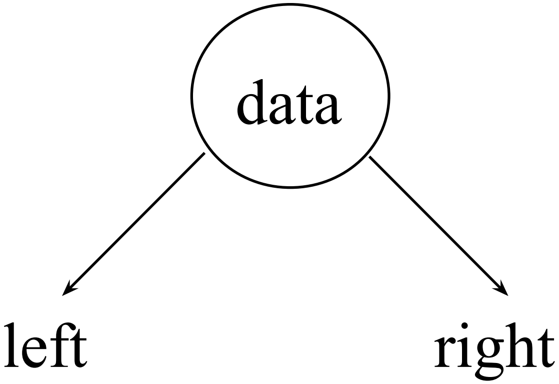
```
struct tnode {
 int data;
 tnode *left, *right;
};
```



left

data

right



data

left

right

Binary Tree
No node has a degree > 2

```
struct TreeNode {
    int          data;
    TreeNode     *left, *right;  // left subtree and right subtree
};

Class BinaryTree {
    private:
     TreeNode * root;
    public:
      BinaryTree() { root = NULL; }
     void add (int data);
     void remove (int data);
     void  InOrder();        // In order traversal
     ~ BinaryTree();
};
```

Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - LRV, LVR, RLV, RVL, VRL, VLR
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - inorder, postorder, preorder

Binary Tree Traversal

In order Traversal (LVR)

```
void BinaryTree::InOrder()      // work horse function
{
    InOrder(root);
}
```

```
void BinaryTree::InOrder(TreeNode *t)
{
    if (t) {
     InOrder(t->left);
     visit(t);
     InOrder(t->right);
    }
}
```

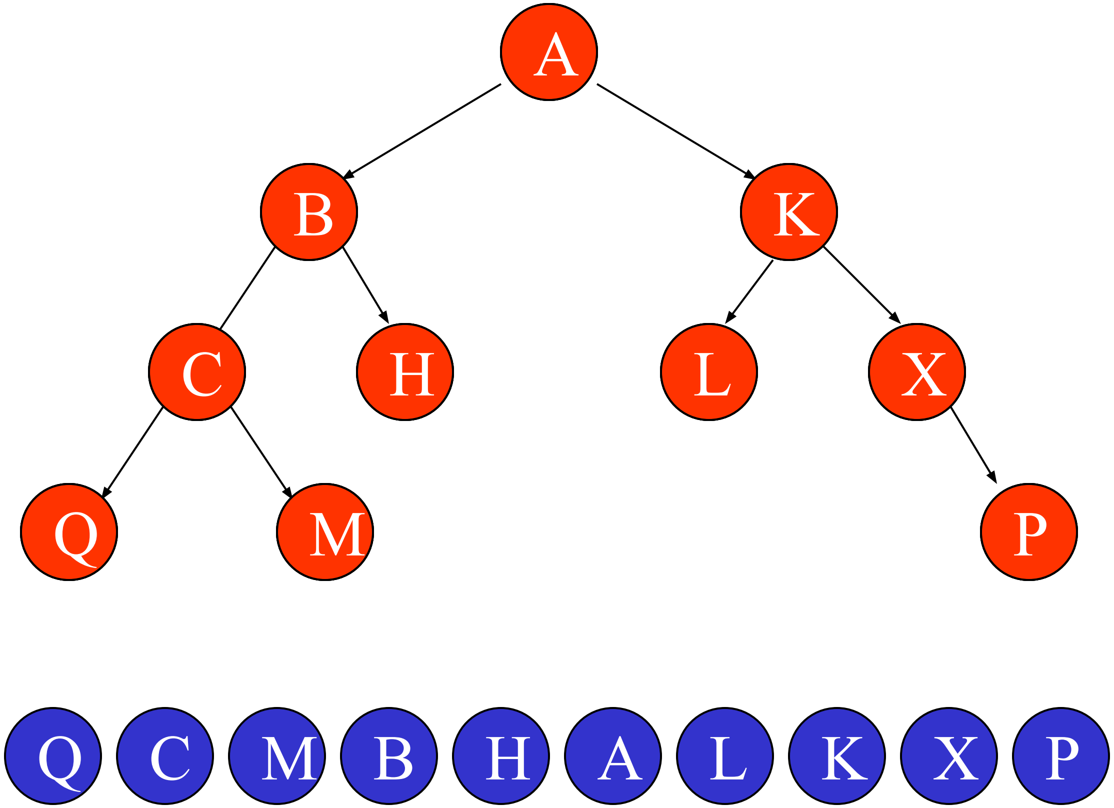
```
void BinaryTree::visit (TreeNode *t) { cout << t->data; }
```

In Order Traversal

- Informally , inorder traversal calls for moving down the tree towards left until you can not go further.
- Then visit the node.
- Move one node to the right and continue.
- If you can not move to right, go back one more node.

Binary Tree Traversal

In Order Traversal (LVR)



```
if (t) {
        InOrder(t->left);
        visit(t);
        InOrder(t->right);
}
```

Binary Tree Traversal

Pre Order Traversal (VLR)

```
void BinaryTree::PreOrder()
{
    PreOrder(root);
}
```

```
void BinaryTree::PreOrder(TreeNode *t) // work horse function
{
    if (t) {
     visit(t);
     PreOrder(t->left);
     PreOrder(t->right);
    }
}
```

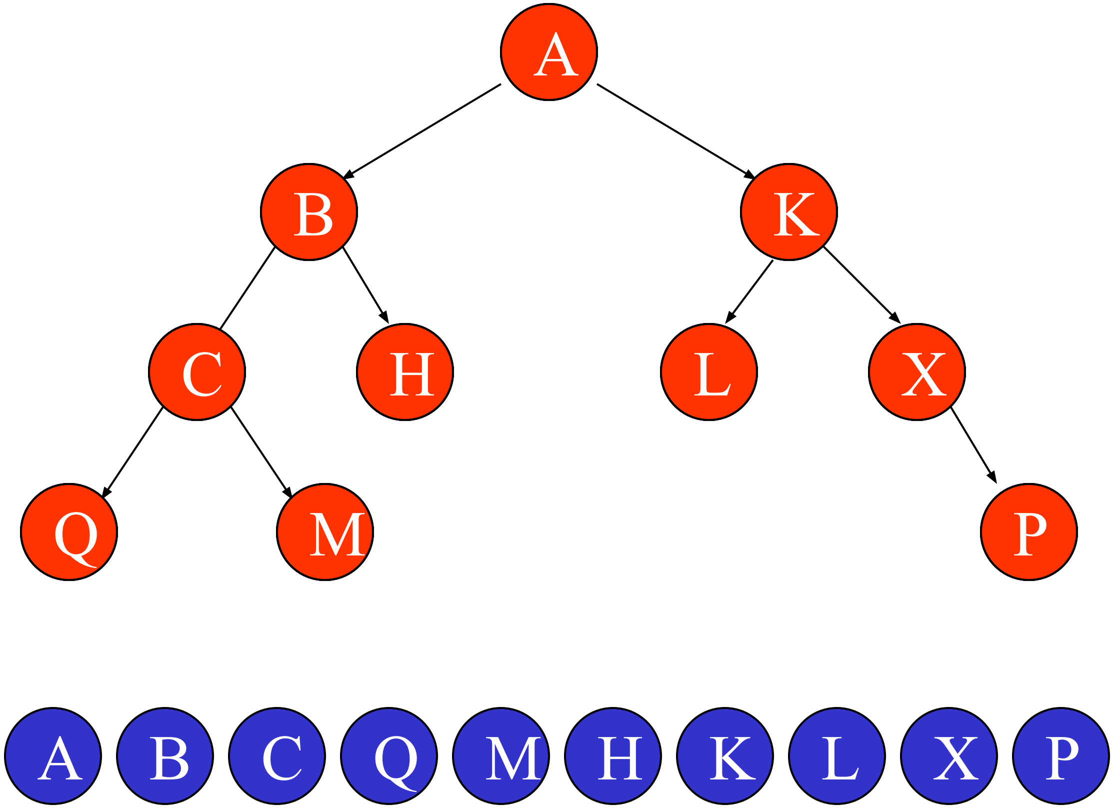
```
void BinaryTree::visit (TreeNode *t) { cout << t->data; }
```

PreOrder Traversal

- In words we say visit the node, traverse left and continue.
- When you can not continue, move right and begin again or move back until you can not move right and resume.

Binary Tree Traversal

Pre Order Traversal (VLR)



```
if (t) {
      visit(t);
      PreOrder(t->left);
      PreOrder(t->right);
}
```

Binary Tree Traversal

Post Order Traversal (LRV)

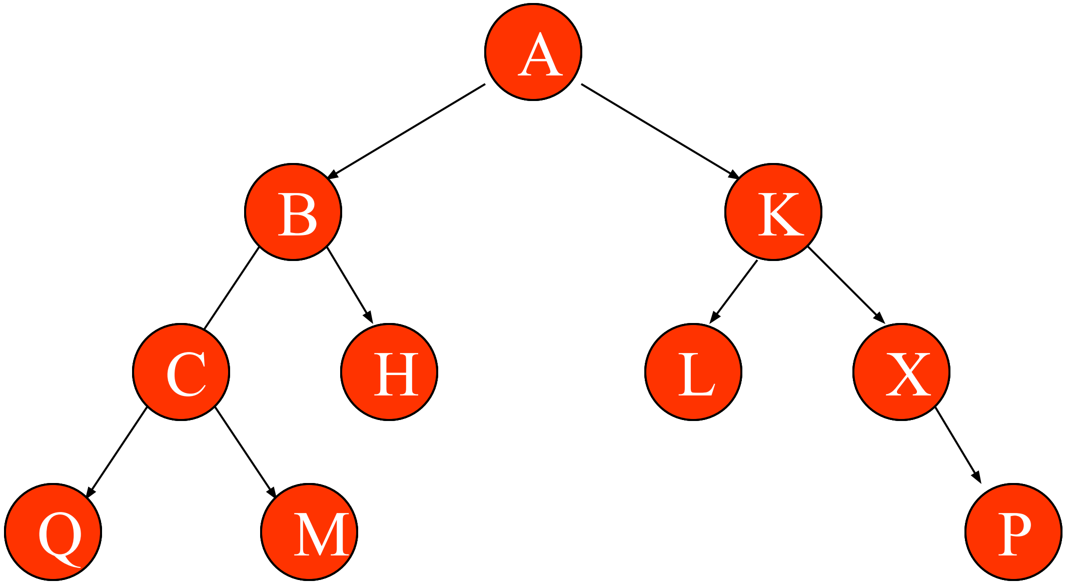
```
void BinaryTree::PostOrder()       // work horse function
{
    PostOrder(root);
}
```

```
void BinaryTree::PostOrder(TreeNode *t)
{
    if (t) {
     PostOrder(t->left);
     PostOrder(t->right);
     visit(t);
    }
}
```

```
void BinaryTree::visit (TreeNode *t) { cout << t->data; }
```

Binary Tree Traversal

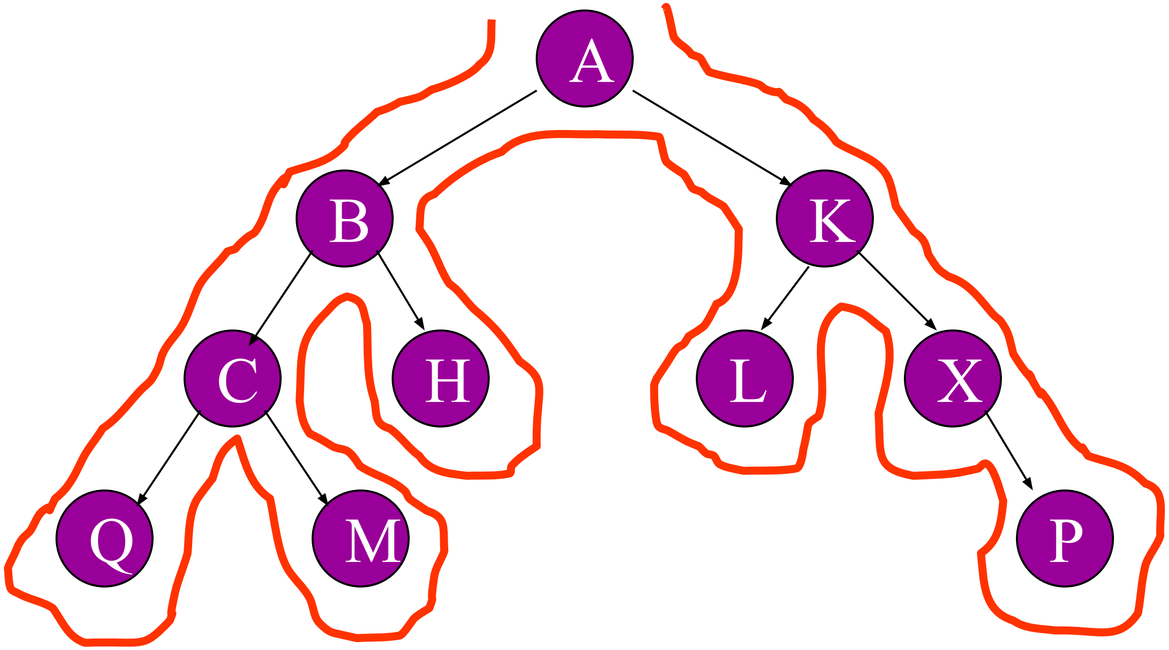
Post Order Traversal (LRV)



Q M C H B L P X K A

```
if (t) {
    PostOrder(t->left);
    PostOrder(t->right);
    visit(t);
}
```

Binary Tree Traversal

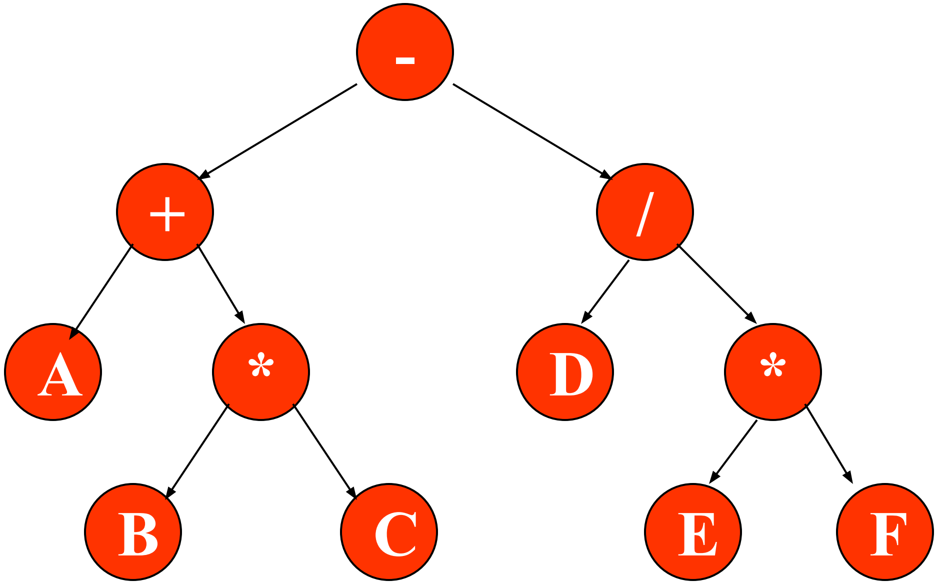


VLR – visit when at the left of the Node **A B C Q M H K L X P**

LVR – visit when under the Node **Q C M B H A L K X P**

LRV – visit when at the right of the Node **Q M C H B L P X K A**

Expression Tree



LVR: A+B*C-D/E*F

VLR: -+A*BC/D*EF

LRV: ABC*+DEF*/-

BinaryTree ::~ BinaryTree();

Which Algorithm?

Delete both the left child and right child before deleting itself

LRV

Non-Recusive Inorder Traversal

```
void Tree::NonrecInorder(){
Stack<TreeNode*> s;
TreeNode *CurrentNode=root;

while(1){
     while(CurrentNode){
          s.push(CurrentNode);
          CurrentNode=CurrentNode□LeftChild;
     }
     if(!s.IsEmpty()){
          CurrentNode=s.pop();
          cout<<CurrentNode□data<<endl;
          CurrentNode=CurrentNode□RightChild;
     }
     else break;
 } }
```